# 堆

## 概述

## 分类

### 二叉堆

### d-堆

d-堆是二叉堆的简单推广，它恰像一个二叉堆，只是所有节点都有d个儿子（因此，二叉堆是2-堆）。

### 左式堆

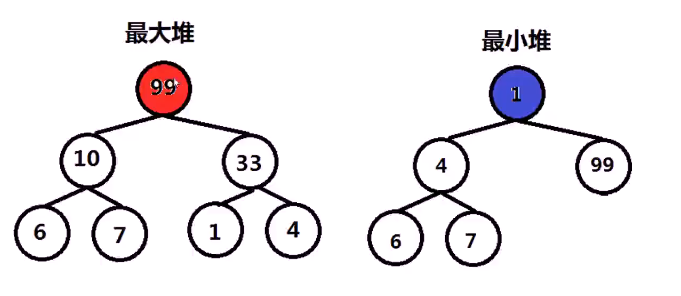
### 斜堆

### 二项队列

## 二叉堆

### 概述

最（大）小二叉堆，最（大）小值先出的完全二叉树。



注：最大堆的左子树、右子树都满足最大堆的定义，最小堆同理。

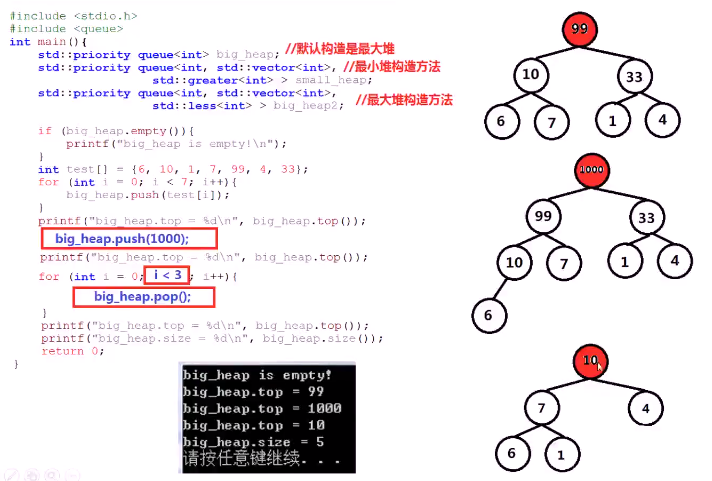
### 表示

### 创建

### 操作

#### 查询

#### 插入



注：掌握最大/小堆的构造。

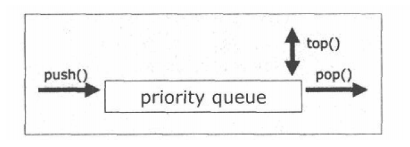
#### 删除

#### 销毁

# 优先队列

## 概述

优先队列priority\_queue其实就一个特殊的queue，其中的元素根据优先级被读取。它的接口和queue非常相近，也就是说，push()对着queue置入一个元素，top()/pop()存取/移除下一个元素，然而所谓的“下一个元素”并非第一个置入的元素，而是“优先级最高”的元素。换言之，priority\_queue内的元素已经根据其值进行了排序。



## 实现

优先队列的实现方法包括：

1. 无序数组实现
2. 无序链表实现

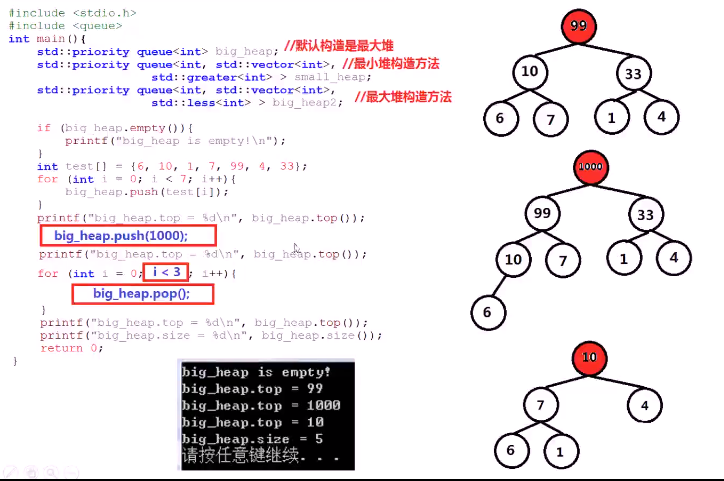
我们可以使用一个简单链表在表头以O(1)执行插入操作，并遍历该链表以删除最小元素，这又需要O(N)时间。另一种方法是，始终让表保持排序状态，这使得插入代价高昂O(N)，而DeleteMin花费低廉O(1)。

1. 有序数组实现
2. 有序链表实现
3. 二叉搜索树实现

二叉查找树对插入和删除操作的平均运行时间都是O(logN)，尽管插入时随机的，而删除则不是。反复除去左子树中的节点似乎损害树的平衡，使得右子树加重。然而，右子树是随机的。在最坏的情形，即DeleteMin将左子树删空的情况下，左子树拥有的元素最多也就是它应具有的两倍。注意，通过使用平衡树，可以把界变成最坏情形的界，这将防止出现坏的插入序列。

1. 平衡二叉树实现
2. 二叉堆实现

二叉堆实现：



## 定义

升序队列：

priority\_queue <int,vector<int>,greater<int> > q;

降序队列：

priority\_queue <int,vector<int>,less<int> >q;

注：greater和less是std实现的两个仿函数（就是使一个类的使用看上去像一个函数。其实现就是类中实现一个operator()，这个类就有了类似函数的行为，就是一个仿函数类了）

### 基本类型

对于基础类型默认是大顶堆：

priority\_queue<int> a;

注：等同于 priority\_queue<int, vector<int>, less<int> > a;

这里一定要有空格，不然成了右移运算符

小顶堆：

priority\_queue<int, vector<int>, greater<int> > c;

### pair优先队列元素

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

int main()

{

priority\_queue<pair<int, int> > a;

pair<int, int> b(1, 2);

pair<int, int> c(1, 3);

pair<int, int> d(2, 5);

a.push(d);

a.push(c);

a.push(b);

while (!a.empty())

{

cout << a.top().first << ' ' << a.top().second << '\n';

a.pop();

}

}

### 自定义类型优先队列元素

#include <iostream>

#include <queue>

using namespace std;

//方法1

struct tmp1 //运算符重载<

{

int x;

tmp1(int a) {x = a;}

bool operator<(const tmp1& a) const

{

return x < a.x; //大顶堆

}

};

//方法2

struct tmp2 //重写仿函数

{

bool operator() (tmp1 a, tmp1 b)

{

return a.x < b.x; //大顶堆

}

};

int main()

{

tmp1 a(1);

tmp1 b(2);

tmp1 c(3);

priority\_queue<tmp1> d;

d.push(b);

d.push(c);

d.push(a);

while (!d.empty())

{

cout << d.top().x << '\n';

d.pop();

}

cout << endl;

priority\_queue<tmp1, vector<tmp1>, tmp2> f;

f.push(b);

f.push(c);

f.push(a);

while (!f.empty())

{

cout << f.top().x << '\n';

f.pop();

}

}

## 接口

优先队列priority\_queue的核心接口主要由成员函数push()、top()、pop()组成：

push()会将一个元素置入priority\_queue中；

top()会返回priority\_queue中的“下一个元素”

pop()会从priority\_queue中移除一个元素

## 应用

优先队列应用如下：

压缩数据：霍尔曼编码算法

查找最短路径：Dijkstra算法

计算最小生成树：Prim树

编写选择算法：寻找第k小的元素。

凡是涉及第k个最值问题，通常可以考虑最大/小堆的实现方式。

### 最小的k个数

**题目：**

输入整数数组arr，找出其中最小的k个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

示例 1：

输入：arr = [3,2,1], k = 2

输出：[1,2] 或者 [2,1]

示例 2：

输入：arr = [0,1,2,1], k = 1

输出：[0]

限制：

0 <= k <= arr.length <= 10000

0 <= arr[i] <= 10000

**分析：**

**方法一：排序**

对原数组从小到大排序后取出前k个数即可。

class Solution {

public:

vector<int> getLeastNumbers(vector<int>& arr, int k) {

vector<int> vec(k, 0);

sort(arr.begin(), arr.end());

for (int i = 0; i < k; ++i) vec[i] = arr[i];

return vec;

}

};

**方法二：堆**

我们用一个大根堆实时维护数组的前k小值。首先将前k个数插入大根堆中，随后从第k+1个数开始遍历，如果当前遍历到的数比大根堆的堆顶的数要小，就把堆顶的数弹出，再插入当前遍历到的数。最后将大根堆里的数存入数组返回即可。

**C++语言中的堆（即优先队列）为大根堆**。

**代码：**

class Solution {

public:

vector<int> getLeastNumbers(vector<int>& arr, int k) {

vector<int>vec;

if(0 == k) //特殊情况：排除k=0的情况

return vec;

std::priority\_queue<int> queue; //使用默认的堆，大根堆即可

for(size\_t i=0;i<arr.size();i++){

if(queue.size() < k)

{

queue.push(arr.at(i));

}else{

if(queue.top() > arr.at(i))

{

queue.pop();

queue.push(arr.at(i));

}

}

}

for(size\_t i=0;i<k;++i)

{

vec.push\_back(queue.top());

queue.pop();

}

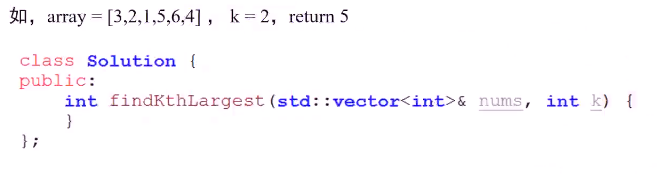
return vec;

}

};

### 数组中第K大的数

**题目：**已知一个未排序的数组，求这个数组中第K大的数字。



注：Leetcode 215

**分析：**

可以采用的方法：对数组进行排序，然后返回第k个数字，这样时间复杂度为O(NlogN)，但是这样不是最优解。

可以采用堆的思想：

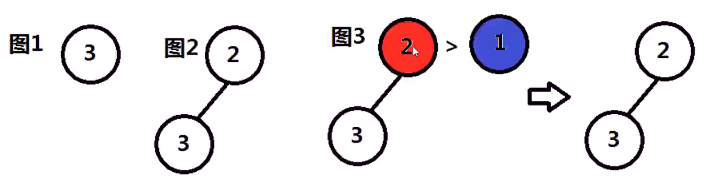
维护一个**K大小**的最小堆，堆中元素个数小于K时，新元素直接进入堆；否则，当堆顶小于新元素时，弹出堆顶，将新元素加入堆。

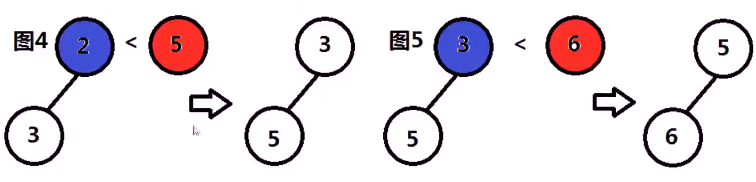
由于堆是最小堆，堆顶是堆中最小元素，新元素都会保证比堆顶小（否则新元素替换堆顶），故堆中K个元素是已扫描的元素里最大的K个；堆顶即为第K大的数。

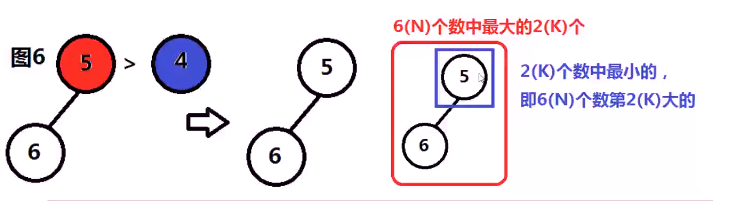
注：这里只需要维护一个k大小的堆即可，不需要将全部数据都存入到最小堆中。

设数组长度为N，求第K大的数，时间复杂度：N\*logK。

如array=[3,2,1,5,6,4]：







**代码：**



注：掌握最小堆的构建方式，即priority\_queue<>

class Solution {

public:

    int findKthLargest(vector<int>& nums, int k) {

        std::priority\_queue<int,std::vector<int>,std::greater<int> > max\_queue;

        for(size\_t i=0;i<nums.size();i++){

            if(max\_queue.size() < k){

                max\_queue.push(nums.at(i));

            }else{

                if(nums.at(i) > max\_queue.top()){

                    max\_queue.pop();

                    max\_queue.push(nums.at(i));

                }

            }

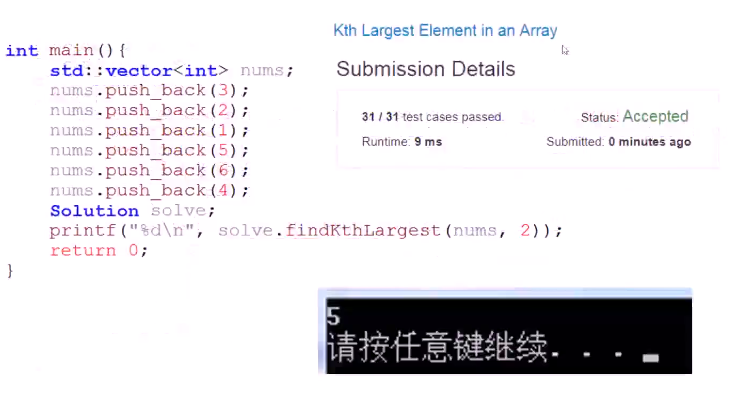
        }

        return max\_queue.top();

    }

};

**测试：**



### 前K个高频单词

**题目：**

给一非空的单词列表，返回前 k 个出现次数最多的单词。

返回的答案应该按单词出现频率由高到低排序。如果不同的单词有相同出现频率，按字母顺序排序。

**示例 1：**

输入: ["i", "love", "leetcode", "i", "love", "coding"], k = 2

输出: ["i", "love"]

解析: "i" 和 "love" 为出现次数最多的两个单词，均为2次。

注意，按字母顺序 "i" 在 "love" 之前。

**示例 2：**

输入: ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4

输出: ["the", "is", "sunny", "day"]

解析: "the", "is", "sunny" 和 "day" 是出现次数最多的四个单词，

出现次数依次为 4, 3, 2 和 1 次。

**注意：**

假定 k 总为有效值， 1 ≤ k ≤ 集合元素数。

输入的单词均由小写字母组成。

**分析：**

有两种基本的思路：

1. 排序

计算每个单词的频率，并使用使用这些频率的自定义排序关系对单词进行排序。然后取前k。

1. 堆

计算每个单词的频率，然后将其添加到存储到大小为k的小根堆中。它将频率最小的候选项放在堆的顶部。最后，我们从堆中弹出最多k次，并反转结果，就可以得到前k个高频单词。

在构造优先队列/堆的时候，可以采用如下的方法：

1. 静态函数

static bool compare(std::pair<string,int> &p1, std::pair<string,int> &p2){

if(p1.second == p2.second)

return p1.first > p2.first;

return p1.second < p2.second;

}

std::priority\_queue<std::pair<string,int>,std::vector<std::pair<string,int> >, compare> queue;

1. 仿函数：重载运算符

struct compare{

bool operator() (std::pair<string,int> &p1, std::pair<string,int> &p2){

if(p1.second == p2.second)

return p1.first > p2.first;

return p1.second < p2.second;

}

};

3、Lambda表达式

注：这种其实就是第k大/小元素的变式，只不过原来构造优先队列的时候只需要存储节点数值和比较大小即可，现在需要存储pair，且比较函数需要自行编写。

**代码：**

class Solution {

public:

    vector<string> topKFrequent(vector<string>& words, int k) {

        struct compare{

            bool operator() (std::pair<string,int> &p1, std::pair<string,int> &p2){

            if(p1.second == p2.second)

                return p1.first > p2.first;

            return p1.second < p2.second;

            }

        };

        std::priority\_queue<std::pair<string,int>,std::vector<std::pair<string,int> >, compare> queue;

        unordered\_map<string,int> mp;

        for(auto w : words)

            mp[w]++; //对应字符的个数+1

/\*这是采用C++11的auto遍历，或采用传统的迭代器方式赋值：

for(int i = 0;i<words.size();i++){

mp[words[i]]++;

}

\*/

        for(auto m: mp)

            queue.push(make\_pair(m.first,m.second));

/\*

for(auto it = mp.begin();it != mp.end();it++){

queue.push({it->first,it->second});

}

\*/

        std::vector<string> s;

        while(k--){

            s.push\_back(queue.top().first);

            queue.pop();

        }

        return s;

    }

};

### 丑数

### 第k个数

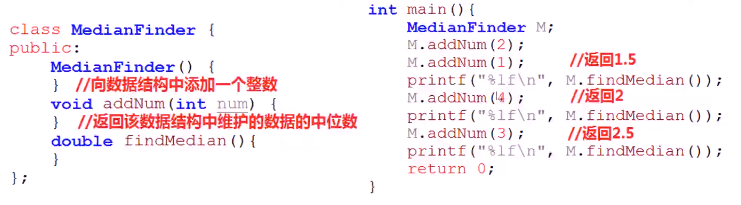
### 寻找中位数

题目：设计一个数据结构，该数据结构动态维护一组数据，且支持以下操作：

1. 添加元素：void addNum(int num)，将整数num添加至数据结构中；
2. 返回数据的中位数：double findMedian()，返回其维护的数据的中位数。

中位数的定义：

1. 若数据个数为奇数，中位数是该组数排序后中间的数。[1,2,3]🡪2
2. 若数据个数为偶数，中位数是该组数排序后中间的两个数字的平均值。[1,2,3,4]🡪2.5



注：Leetcode 295

**分析：**

最直观的方法：

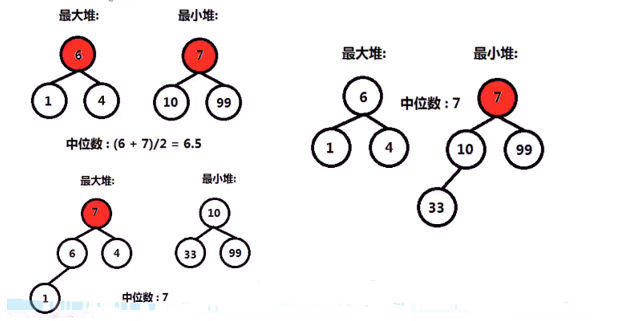
存储结构使用数组，每次添加元素或查找中位数时对数组排序，再计算结果。

时间复杂度：

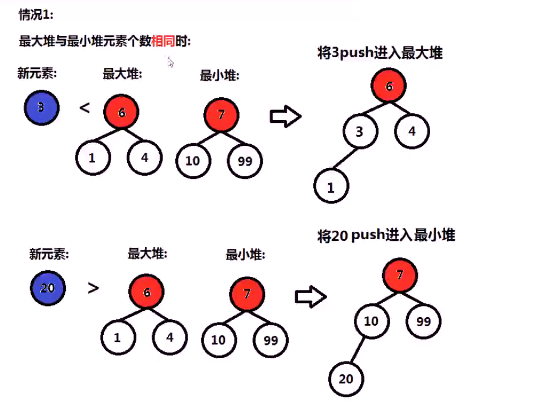
1. 若添加元素时排序，addNum复杂度O(n)，findMedian复杂度O(1)
2. 若查询中位数时排序，addNum复杂度O(1)，findMedian复杂度O(nlogn)

若添加元素或查询中位数是随机的操作，共n次操作，按照上述思想，整体复杂度最佳为O(n^2)。

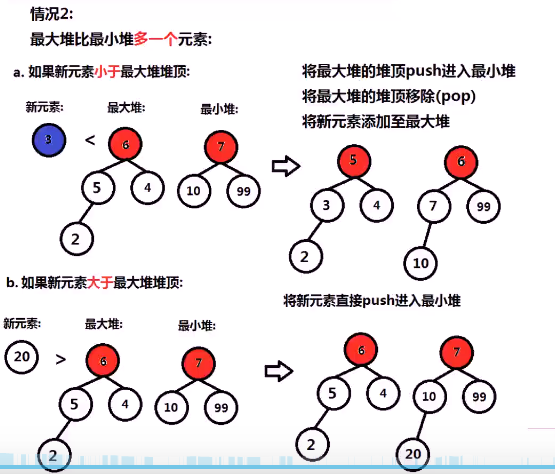
除了上述方法，还可以动态维护一个最大堆和一个最小堆，最大堆存储一半数据，最小堆存储一般数据，维持最大堆的堆顶比最小堆的堆顶小。



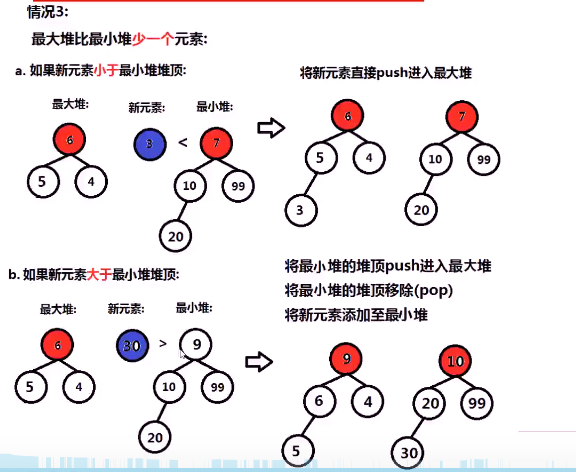
情况1：最大堆与最小堆元素个数相同时：

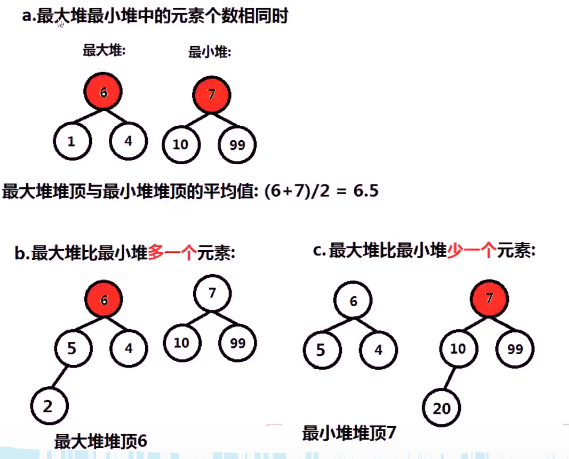


情况2：最大堆比最小堆多一个元素：



情况3：最大堆比最小堆少一个元素





**代码：**

